Mission Interactive Scenario Studio for Autonomous Spacecraft

Dr. Thomas Starbird,
Mr Imin Lin,
and
Mr. Adans Y. Ko

Jet Propulsion Laboratory
4800 Oak Grove Dr.
Pasadena, CA 91109
818-354-1033
Thomas.W.Starbird@jpl.nasa.gov

Abstract— We propose building a Studio enabling the use of diverse existing mission activity and scenario patterns, the creation of new ones, and the modeling of their effects using existing modeling tools. The core of the Studio is a component-based Type Library, which captures years of mission adaptation patterns in various forms.

The Studio works as a content server to capture the developed adaptation knowledge for reuse and provides bridging into different mission uplink implementations, including the Mission Data System [1] (MDS) state/goal machinery. Various activities can be coordinated, controlled, and reused through the Studio's component interface to establish and model a mission scenario. A special component Factory mechanism will be in place to facilitate the adaptation of projects into the Studio.

The architecture of the Studio reflects and enforces a division of knowledge and actions into three parts: Model, Controller, Viewer. The Model contains information about (a proposed version of) the spacecraft and mission. The Controller contains logic for constructing scenarios of mission activities. The information in the Model and Controller is principally in the form of reusable patterns. A Viewer can be a simple or complex software system. For example, Apgen [2,3] is one possible viewer, MDS is another.

A 3-tiered infrastructure is used for the Studio, reflecting the Model, Controller, Viewer architecture[4,5].

The Studio is useful in pre-phase A of a project by enabling spacecraft design options to be played against desired mission scenarios. In later design phases of a project, the construction and modeling of more detailed scenarios is supported by the Studio.

1. INTRODUCTION

In the early life of a project, before the spacecraft or mission has been designed, design options are studied. Variations of spacecraft design are considered, as are variations in the overall mission. Models at various levels of fidelity are used to derive implications on cost, science return, and other key features from each variation. Some aspects of intended operation after launch are relevant. For example, a spacecraft design implies how much power is available to the spacecraft, which can constrain the way the spacecraft is operated. Conversely, operational requirements, such as the number of images to be taken, yield constraints on spacecraft design. Hence it is useful to model aspects of intended operation.

Prototypical "days in the life" of the mission can be developed to draw implications from and on spacecraft and mission design. Each scenario consists of activities that are modeled as to their effects on key resources, often power, energy, telecommunication data rates from the spacecraft to earth, and data volume of onboard storage and downlink. Frequently a scenario consists of a handful of basic activities each repeated many times, perhaps with variations.

As mission and spacecraft variations are considered, scenarios are added and changed, and facets of resources are changed. For example, changing from a solar-powered spacecraft to a nuclear-powered one changes the amount of power available, and the timing of availability. So the models of resources change, as do pertinent scenarios.

Executing a scenario against the models of resources can confirm or deny the viability of the scenario. Unviable ones can be changed to become viable, yielding limits on durations or other aspects of activities. Mission performance can be estimated by incorporating into the models parameters considered indicative, including even some rough estimates of "science value". Such parameters can be useful in comparing one mission variation to another even if the absolute value of the parameter has no intrinsic meaning.

As the mission progresses from studying grossly different options to questions of finer scale, the scenarios and resource models can fruitfully be made more detailed. There is a natural progression, aligned with the Mission Data System (MDS) state analysis process, of identifying and codifying lower level resources (MDS state variables). The further development of scenarios is aligned with the MDS process of identifying goal types. (Note: MDS, under development at JPL, is a software framework for future space missions.)

The MISS facilitates the construction of activities, scenarios, and models of resources. One key ingredient is the use of "patterns" for specifying these individually and in related groups. The use of patterns eases the construction of variations. Another key ingredient is software "components". Components can represent different versions of a resource model, for example, that can be swapped one for another. This enables the user of MISS to piece together a tailored modeling system. Sometimes it is even useful for components to be swapped during the course of a modeling execution.

This paper describes the vision for the MISS, and relates a case study that is using the concepts, though often in a manual mode. The case study is the Mars Science Laboratory (MSL) project, which is a spacecraft that is to be launched in 2009. The mission includes an analytical laboratory on the surface of Mars.

2. MISS SYSTEM ARCHITECTURE

The architecture has three parts: Model, Controller, and Viewer.

2.1 Model

The Model specifies information about the spacecraft and mission that has existence independent of scenario construction. In particular, the Model specifies the design of the spacecraft, including, for example, the devices that comprise the spacecraft, the possible states of each device, the spacecraft resources, and the relation of resources to states (e.g., how much power a hazard avoidance camera uses when in the "sampling" state). The Model also contains mission activities or goals that will form the building blocks of scenarios.

The Model need specify only those aspects that will affect the construction or analysis of scenarios, and only to the detail required for such. In pre-phase A of a mission, where trade studies are comparing different possible designs of the spacecraft, each candidate design has a different version of the Model.

Much of this kind of data is available elsewhere during the design of a mission, so the Studio imports it from those sources.

The Model portion of the Studio is patternized; the Model's information is structured into reusable patterns. For example, there is a device pattern, which abstracts the notion that a device has several possible states. Examples of patterns are given later in this paper.

2.2 Controller

The Controller has information needed for constructing a mission scenario from the Model. A mission scenario is a collection of activities to be accomplished during a given span of time during the mission. The activities include timing information and their effect on states of devices in the spacecraft.

The Controller portion of the Studio is also patternized. Each pattern specifies an aspect of constructing a scenario. For example, there would be a pattern (several actually) for merging two timelines. The Controller patterns refer to Model patterns.

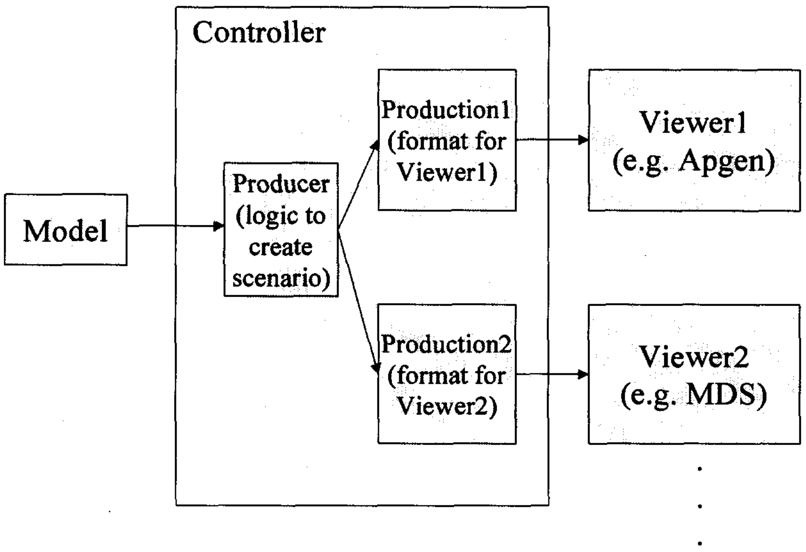
Controller
Model
Producer (logic to create scenario)
Production1 (format for Viewer1)
Production2 (format for Viewer2)
Viewer1 (e.g. Apgen)
Viewer2 (e.g. MDS)

Figure 1: Studio Architecture

To construct a scenario involves instantiating many Model patterns (for example, constructing (models of) several devices on the spacecraft that will be used in the scenario), and instantiating a Controller pattern that refers to those models. Instantiating a pattern can actually cause more than one pattern to be exercised, since the definition of a pattern can refer to other patterns. In the first version of the Studio, patterns are created manually. In future versions, the Controller will create not only scenarios, but also models, built from existing models and stored back into the Model.

The Controller is the heart of the Scenario Studio. Whereas the Model contains mostly information that is more general than needed for scenarios, the business of the Controller is specific to scenarios. The Controller contains the "business logic", i.e., the logic specific to the construction of scenarios.

2.3 Viewer

A Viewer is a software system that ingests the scenario constructed by the Controller. The Studio will use existing software systems as Viewers, such as tools that display timelines with values computed by modeling the effects implied by a scenario.

2.4 Relation of Viewer and Controller

The Controller is divided into two parts: the Producer and the Production. The Producer contains the logic that is independent of the Viewer; this is the central core of the Controller. The Production contains information needed to format the output scenario suitable for ingestion by the desired Viewer.

3. SYSTEM ARCHITECTURE

3.1 Patterns

A pattern is a group of reusable assets that can help to speed up the process to create and deploy a new Ground Data System (GDS) for Missions.

The patterns leverage on the experience of JPL Mission System Engineers / Architects to create new GDSs in a more effective and efficient way. There are static and dynamic patterns. These patterns are based on Mission requirements, then quickly translated through the different levels of pattern assets to identify a final solution design and product mapping appropriate for the application being developed.

3.2 Static Patterns

The Static Pattern (see figure 2) consists of the following levels [6]:

1.Mission Project Patterns: They come from Mission requirements (a.k.a. Customer requirements), and represent different Mission types (e.g., Orbital, In-Situ, Flyby, etc.).

2. Business Patterns: The next level of Mission requirements provides us with a set of Mission and Spacecraft specifications, which usually describe what are the components / subsystems of the spacecraft, the behavior of the subsystems, the control flow and the data flow of each subsystem. The Mission Operational concept provides us the requirements on users' interactions with the mission operations' and the ground data system's applications and their data products. The knowledge of the business patterns usually are captured into software models (e.g, Data Storage model, Power subsystem model, Attitude Control subsystem model, and Telecom model).

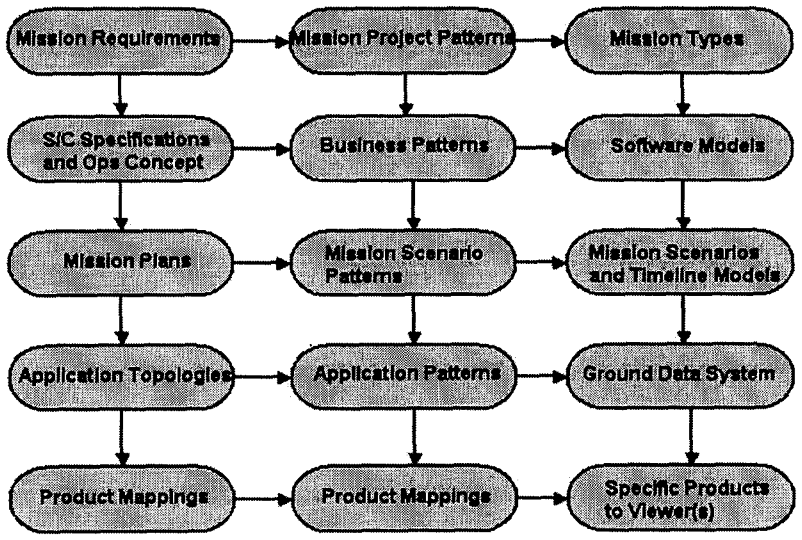

Figure 2. Static Patterns

3. Mission Scenario Patterns: This level of patterns is based on the Mission Plan. Depending on the mission type, the scenarios from the mission plan will be patternized. Examples are Data Downlink communication scenario, Rover Drive over a Sol scenario, Pan Cam Imaging scenario, and S/C occultation scenario. The pattern definitions of these scenarios are captured into the Mission Scenario and Timeline models.

4. Application Patterns: They represent the partitioning of the application logic and data together with the styles of interaction between the logic tiers. We are proposing to use an "N tiers" Architecture [5] with Web Server, Application Server, and Database Server (see figure 3) for the MISS system architecture. The Model layer usually is an essential part of the system. It contains the definitions of the mission scenarios and models. Some of the models are created as a component object or a dynamic library. The model layer is also called "data layer". It uses a LDAP server for managing the name space and for faster access. Mission scenario and model patterns are stored in this layer. The patterns are created from the results of system engineering of the mission design, subsystem behaviors, and also the mission operations concept. The Controller layer, also called "applications layer", has the logic of the scenarios and models production. It uses XML, XSTL for managing the input, webmacro templates or Java Beans to handle specific

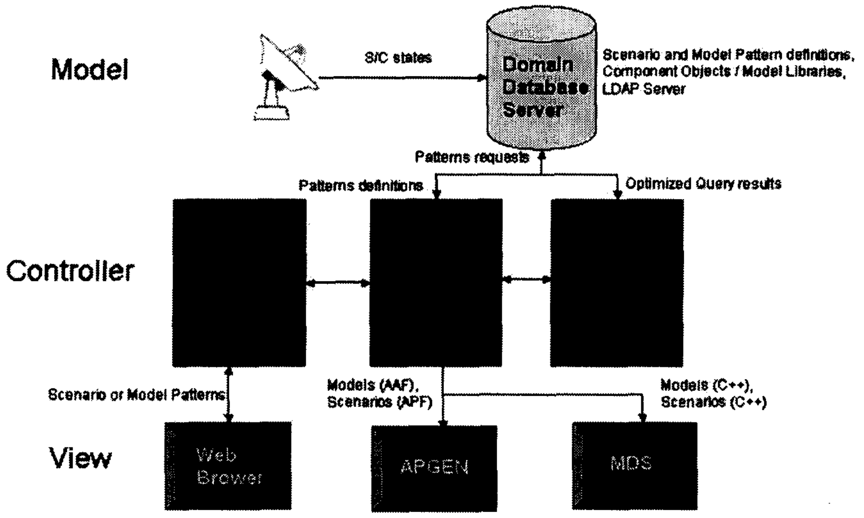
Model
S/C states
Domain Database Server
Scenario and Model Pattern definitions, Component Objects / Model Libraries, LDAP Server
Patterns requests
Patterns definitions
Optimized Query results
Controller
Scenario or Model Patterns
Models (AAF), Scenarios (APF)
Models (C++), Scenarios (C++)
View
Web Brower
APGEN
MDS

Figure 3: N-Tier Architecture: Web, Application, and Database

functionalities (e.g., pattern request handler, I/O interfaces, database access, and presentation output for viewer). A search engine is also included in the controller layer, in order to optimize the query of the database server. The view layer is where the scenarios and models are in a particular format able to be viewed. The controller has the knowledge to publish the appropriated content which depends on the specific viewer (e.g., Apgen, MDS, or web browser).

5. Product mappings: The product mappings will use the "push" or "publish" method to populate the solution into the different viewers (see figure. 3).

3.3 Dynamic Patterns

The Dynamic Pattern [6] (see figure 4) introduces two more pattern concepts: integration patterns and composite patterns. The Integration Patterns connect other Business patterns together to create applications with advanced functionality. While the models from the Business Patterns represent the S/C behavior model and the Mission Scenario Patterns represent the Mission Scenario definitions, the integration patterns will have the Mission and Flight rules, preference selections, exclusiveness, persistence, and invalidation checks that govern the composition of different mission scenarios or mission timelines. The Controller translates interactions with the view into actions to be performed by the definitions of the Integration Patterns and the models from the Business Patterns. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view. The Composite Patterns combine Business patterns and Integration patterns to create complex, advanced GDS. (We are not addressing the composite patterns any further in this paper.)

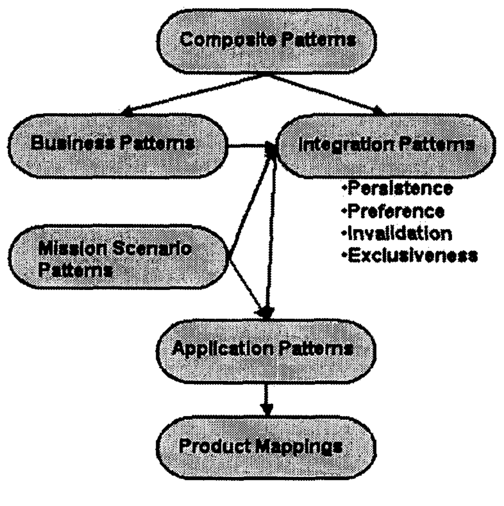

Figure 4: Dynamic Patterns

3.4 Pattern Instantiation

Patterns in general are processes that can be abstracted. This abstraction can occur at different phases of a mission design. When a pattern is applied, normally an instantiation step(s) is taken to turn this abstraction into a concrete entity that can be visualized. In fact, the MCV model itself is a pattern adapted for mission planning as will be presented in the following sections.

Patterns can be archived and searched and composed to form new patterns. The searching and composing are not discussed in this paper.

A pattern should have the following minimum elements:

- Description
- Keyword(s)
- Instantiation parameters
- Presentation(s) per controller per viewer
- Producer per controller

As we will see in the following section, pattern instantiation is the code of the Controller function. The system model request is sent to the Controller. The Controller locates a Producer that can handle this request. The Producer selects the right Presentation(s) and passes the request in the format that fits the Presentation. The Presentation then produces the outputs for the Viewer.

3.5 Case Study: Timeline Scenario

A Timeline is a time based sequence of events. Other than some system administration events, most events are records of changing of certain system state(s), for example, set the Heater state to "On" at 20 seconds from the start of the Timeline. Another example is: trigger an onboard procedure such as "Starting the Traversing at 30 seconds from the start of the Timeline". The mission adaptor can use Timeline to capture most operational sequences into a collection of Timelines. Multiple Timelines can be composed into a single Timeline. In this Timeline scenario, two patterns were defined:

State transition – Spacecraft consists of devices. Each device has states. An operational sequence can be thought as a sequence of device change states.

Timeline composing – When composing multiple timelines into a single Timeline, the following rules shall be carried:

- Redundant state
  If the new state of a device is a redundant state to the current state, the new state change is ignored. For example, the Heater state "Standby" is a redundant state of the state "On".
- Preferred state
  If two state changes occur at the same time, the less preferred state is ignored. For example, if the Heater is requested to be "On" and "Off" at the same time, "Off" is ignored.
- Invalid state
  If an invalid state is requested, an error message is generated and the new state is ignored. For example, an "On" state is requested without a "Standby" state occurring first.
- Persistent state
  Persistent state is easier to explain with an example. First state change: at time of 2 seconds, set the Heater to "On" and at time of 7 seconds, set the heater to "Off". Second state change: at time of 2.5 seconds, set the Heater to "On", at time of 6 seconds, set the Heater to "Standby". The end result is: at time of 2 seconds, set the Heater to "On", at time of 7 seconds, set the Heater to "Standby". This example implies a preference that "On" > "Standby" > "Off".

Of course, this Timeline composing pattern is missing a few important behaviors. One of the missing behaviors is Exclusiveness. For example, two requests to set Heater "On" is not a problem. But setting a Camera to "Sampling" in the same interval of time is a problem.

3.6 Static pattern, Dynamic pattern

One way to differentiate a Static pattern and a Dynamic pattern is the type of implementation of the pattern. Normally, the Static pattern consists of a set of data items. State transition is a Static pattern. A dynamic pattern normally consists of procedures. Timeline composing is a Dynamic pattern. As we will demonstrate in MISS's 3-tiers infrastructure, the Static pattern is implemented as Oracle data tables, while the Dynamic pattern is implemented as a Java bean component which can be loaded and executed on demand.

3.7 Implementation

In this case study, MISS is producing a Timeline for the DSMS Apgen tool. The MISS/Apgen

Control/Production is implemented as a template in Webmacro format. (http://www.webmacro.org). The Model and Control/Producer are implemented in the Timeline Java bean (see Figure 5). In the future, these two shall be separated, as we will discuss later.

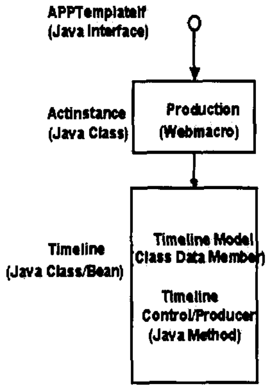
APPTemplateIf (Java Interface)
ActInstance (Java Class)
Production (Webmacro)
Timeline (Java Class/Bean)
Timeline Model Class Data Member)
Timeline Control/Producer (Java Method)

Figure 5: Model Timeline Production

3.8 Model

Model is where the mission data is kept.

Sample Timeline Model entry 1:

```
{"PSE","-1","00:00:00","24:36:00",
new Integer (StateChange.FIXED).
toString(),"On","0","Null","3",
"GG_SOL","0"},
```

```
Device type: PSE
Which device: -1 means the first
      PSE of all the PSE's.
Start time: 00:00:00
Duration: 24:36:00. Duration is
      meaningless unless the next
      field is DERIVED.
Stop state significance: FIXED.
      Insignificant
Start state: "On"
Last Parameter: 0 (future use)
End State: Null
Repeating count: 3
Repeating interval: 1 SOL
Repeating starting index: 0
```

This Model info is saying: Set the first PSE device to "On" at 00:00:00 of a SOL. Repeat the setting for 3 SOLs, starts from SOL 0.

Sample Timeline Model entry 2:

```
{"PSE","-1","00:00:00","00:36:00",
new Integer (StateChange.DERIVED).
toString(),"Standby","0","On","3",
"GG_SOL","0"},
```

The difference between Sample 2 and Sample 1 is the significance of the End state. In Sample 2, the End state is significant. The Model info is saying: Set the first PSE device to "Standby" at 00:00:00 of a SOL. After 00:36:00, set it to "On". Repeat the setting for 3 SOLs, starts from SOL 0.

The reason the Model info is implemented as Java data is only for quick prototyping. In our next paper, we will talk about the using of Aspect-Oriented Programming(AOP) and Object-Oriented Data Base (OODB) as the true implementation.

3.9 Control

Control is where the business processing of the Model take place. The result of the processing is presented through a use of Production based on the Viewer.

3.10 Production

The viewer in this case study is Apgen. The following is a Webmacro template of the Apgen AAF and APF.

```
#foreach $(StateChange) in $(DB.Devices)
{

#foreach $(StateRepeat) in $(StateChange.StateRepeats)
{

activity instance $(StateChange.Name) of type
$(StateChange.TypeName) id

$(StateChange.Name)
begin
        attributes
            "Start" = GG_SEQ_START_TIME +
$(StateChange.Elapsed_Time) +
                      $(StateRepeat.Elapsed_Time);
            ##"Duration" = $(StateChange.Duration);
            "Duration" = GG_SET_ACT_DUR;
        parameters
            (#foreach $(Parameter) in
$(StateChange.Parameters)

{$(Parameter.Value),}$(StateChange.LastParameterValue));
    end activity instance $(StateChange.Name)
#if ($StateChange.HandleOffState)
{
activity instance $(StateChange.Name) of type
$(StateChange.TypeName) id

$(StateChange.Name)
    begin
        attributes
            "Start" = GG_SEQ_START_TIME +
$(StateChange.Elapsed_Time) +
                      $(StateChange.Duration) +
$(StateRepeat.Elapsed_Time);
            ##"Duration" = $(StateChange.Duration);
            "Duration" = GG_SET_ACT_DUR;
        parameters
            (#foreach $(Parameter) in
$(StateChange.Parameters)

{$(Parameter.Value),}$(StateChange.TerminateParameterValue));
    end activity instance $(StateChange.Name)
}  ##End of if ($StateChange.HandleOffState)
} ##End of foreach $(StateRepeat) in
(StateChange.StateRepeats)
}##End of foreach $(StateChange) in $(DB.Devices)
```

For the readers who do not know about Apgen, it is not required to know Apgen before we can show you what is in this Production. Production is a material that can guide the Producer to produce a run stream for a particular Viewer. In this Production, $([id].[id]) is an indication of parameter(s) required to materialize this Production. Let us use the Sample Timeline Model Entry 2 as example:

```
{"PSE","-1","00:00:00","00:36:00",
new Integer (StateChange.DERIVED).
toString(),"Standby","0","On","3",
"GG_SOL","0"},
```

StateChange.Name->"PSE",
StateChange.Elapsed_Time->00:00:00
StateRepeat.Elapsed_Time->GG_SOL

```
StateChange.LastParameterValue
                     ->"Standby"
StateChange.TerminateParameterValue
                     ->"On"
StateChange.StateRepeats
                     ->StateRepeat[3]
```

The last one is of the most interesting. It basically saying, StateRepeats is an array of 3 of StateRepeat. The StateRepeat object, then carries the StateChange repeating info as shown above.

The Webmacro provides us with a way of separating the processing data from the viewing of the data. Imaging if the template is an HTML based, than it simply created an web HTML for the same data. In our next paper, we will use this approach to create C++ code that can be compiled by a C++ complier and executed.

3.11 Producer

The Producer is the process that implements the "business logics". Section 2.3, we have defined the "Business Logics" for Timeline composing. The Producer of Timeline is a Java bean. Segments of the code are listed and discussed.

```
int nbrdevices = 0;
int currentdeviceindex = 0;

for (int totalst = 0; totalst <
TimelineStateChange.length;totalst++) {
    int currentcount =
Integer.parseInt(TimelineStateChange[totalst][1]);
    if (currentcount < 0) currentcount = (-currentcount);
    Integer.parseInt(TimelineStateChange[totalst][1]);
    nbrdevices = nbrdevices + currentcount;
}

stchg = new StateChange[nbrdevices];
```

The above code is to set up all the State changes in a Timeline. See the line "stchg = new StateChange[nbrdevices]" creates an array of StateChange beans. Each instance of bean represents an state change event in the Timeline.

```
boolean moredeploy = true;
while(moredeploy) {
    Current_State=((StateDeploy)Timeline.getLast()).getState();
    Current_State_Holder=(State)States.get(Current_State);
    getNextSelectedDeploy();
    if (Current_State.compareTo(NextSche.getState()) != 0) {
        //Check if the next state is a valid state
        Current_State_Holder.setNext_State(NextSche.getState());
        if (Current_State_Holder.isNext_State_Valid()) {
            //Check if the next state is redundant
            if(!Current_State_Holder.isNext_State_Redundant()) {
                //Now check for Preference persistance
                if  (!isLowerPreference((StateDeploy)
                              Timeline.getLast(),NextSche))
                    //Put this into Timeline, else drop it
                    Timeline.addLast(NextSche);
                else
                    combineRedundantDeploy ((StateDeploy)
                                        Timeline.getLast(),NextSche);
            }
            else
                combineRedundantDeploy((StateDeploy) Timeline.getLast(),
                                                       NextSche);
        }
        else
            //Logging invalid state error
    }
    else {
        //Same state, but check the duration and determine if it is needed
        //to extend the original request
        combineRedundantDeploy( (StateDeploy)Timeline.getLast(),NextSche);
    }
    //continue as long as the finalDeploy is still on the Workline
    if (!Workline.contains(finalDeploy))
        moredeploy = false;
    else
        //Pick the first one on the Workline
        NextSche=(StateDeploy)Workline.removeFirst();
    }
}
```

This code segment show the Time composing algorithm. It first check for the redundancy. If the new state is a redundant state, it calls *`combineRedundantDeploy() to do the persistency check.`*

3.12 Studio

To people who know about template programming, this may appear to be a template instantiation process using Webmacro java classes. (For another java template, see http://jakarta.apache.org/velocity.) But template instantiation only represents one controller implementation, that used for MSL trade study. The number of possible implementations is unlimited.
Another popular implementation involves using XML/XSLT. In this approach, the Webmacro template is replaced by an XSLT translator, which "translates" the incoming message into an Apgen control stream. The selection of the controller implementation depends on the ease and performance.
The question comes to mind, how are all these Producers, Productions saved? Producers in MISS are Java beans. Productions are basically text files. In the case study, the Model is implemented as a data member. But it really should be a procedure (Java bean) to access a backend data base. It takes all these pieces to make one service. All these strongly suggest a

web services infrastructure is the best implementation of MISS.

In our next paper, we will discuss how to deploy MISS as web services.

4. CONCLUSION AND FUTURE WORK

Mission planning is a very complex and time consuming process. Because of its complexity, we believe this expert system approach with reusable patterns is a viable solution.
The future work involves the following:

- Pattern saving and searching.
  Currently all the Productions are Webmacro text files. There is no indexing among them, and no content keywords are exposed. This makes keyword searching difficult. A solution under consideration is to incorporate a content data base for easy saving and searching.
- Pattern composing.
  Need to develop a formal specification describing a pattern. A pattern in a way resembles a component such as a Java Bean or Window COM object, in which patterns have properties that describe how they can be composed. This is one of the main topics of our next paper.
- MISS as a web service.
  A web service provides easy access for the users. A web service also provides a good infrastructure that combines different programming tools into a cohesive application. With the advancement of the web application server, quite a lot of needed utilities are ready to use. Most importantly, a commercial search engine can be applied to facilitate the pattern searching [7]
- Studio.
  MISS is designed to be a multi-mission tool, in that project structure is required. Each project can have its own "my place". Each project can create its own patterns and share with others. Version control of patterns becomes important. Currently the MISS team is reviewing JMX (Java Management Extension) with JBOSS [8] and Sniff+ from WindRiver. (http://www.windriver.com/products/sniff_plus).

5. REFERENCES

[1] Dvorak,D., Rasmussen, R., Reeves, G., and Sacks, A. Software Architecture Themes in JPL's Mission Data System. Proceedings of the 2000 IEEE Aerospace Conference, Big Sky, Montana, March, 2000

[2] Maldague, P., Ko, A.Y., Page, D.N. and Starbird, T.W., 1997, "APGEN: A Multi-Mission Semi-Automated Planning Tool", in First International Workshop on Planning and Scheduling for Space Exploration and Science.

[3] Maldague, P., Ko, A.Y., "JIT Planning: an Approach to Autonomous Scheduling for Space Missions", in March, 1999, IEEE Aerospace Conference, Aspen, Colorado

[4] "The Model-View-Controller Architecture", http://rd13doc.cern.ch/Atlas/Notes/004/Note004-7.html. As a reference for our MCV model

[5] "Model View Controller", http://www.objectarts.com/EducationCentre/Overviews/MVC.htm

[6] "Patterns for e-Business: A Strategy of Reuse" ,Johnathan Adams, Srinivas, Guru Vasudeva, and George Galambos

[7] "Build a Smarter Search Engine", by Baylor Wetzel. Java Pro October 2002

[8] "JBoss Administration and Development", The JBoss Group. "Patterns and Software: Essential

6. BIOGRAPHY

__Dr. Thomas Starbird,__ is a Principal in System Design in the Mission Systems Engineering Section of the Jet Propulsion Laboratory (JPL). He received a B.A. from Pomona College, Claremont, California, and a Ph.D. from the University of California at Berkeley, both in Mathematics. He has participated in software development and in Mission Operations System development for several space projects at JPL, including the Galileo Project, where he was the Ground Software System Engineer for several years before launch. More recently he has led the Mission Planning and Execution portion of the Mission Data System, a software framework for future flight and ground systems.

__Mr. Imin Lin__, is a Program Element Manager in Multi-mission Ground Data System tools, which includes mission planning, sequence generation, and S/C telecom analysis for current and future JPL missions and the DSN. He got a B.S. in Physics and M.S in Computer Science

__Mr. Adans Y. Ko__ is a Software Systems Design Engineer and Technical Group Supervisor in Section (314) at JPL. He is responsible for the Multi-mission Ground Data System tools, which includes mission planning, sequence generation, and S/C telecom analysis for current and future JPL missions and the DSN. He received NASA's Exceptional Service Medal for his work on Voyager's onboard Computer Command Subsystem for mission to Uranus and Neptune. He was the Development Manager of the Mission Planning and Sequence Subsystem. He got his B.S.C.S. degree from the Utah State University, Logan, Utah. in 1982, his M.B.A. degree from University of California, Los Angeles in 1993.